CSN08x14

Scripting for Cybersecurity and Networks

Lecture 5: Complexity of Algorithms; Timing Python code

Today's Topics

You will learn about:

- Functions(in mathematical sense)
- Growth of functions
- Complexity of algorithms
- Hash tables
- Timing code with Python
- Tuning Python code → next week
- Python plots: pyplot
- Modules: time, numpy, matplotlib

Some terms we will use:

- big-O (big-Ω, big-Θ)
- Linear growth
- Quadratic, Cubic, Polynomial growth
- Logarithmic, Exponential growth

Go to www.menti.com

code **xxxx**

Comparing searching Algorithms

Comparing search algorithms

Looking up numbers in a phone book
How many accesses are needed to find a record?

(1K ~ 1000 ~ 2^{10})

(1G ~ 1,000,000,000 ~ 2^{30}) etc

Phone book size	Linear search	Binary search
1K		
1M		
1G		
1T		

Comparing search algorithms

Looking up numbers in a phone book
How many accesses are needed to find a record?

(1K ~ 1000 ~ 2^{10})

(1G ~ 1,000,000,000 ~ 2^{30}) etc

Phone book size	Linear search	Binary search
1K	512	10
1M	524,288	20
1G	536,870,912	30
1T	549,755,813,888	40

Comparison

(n is the number of values in the list i.e. the length of the list or array)

Linear search

- Requires n/2 steps on average
- n steps if search value not in list*
- List need not be sorted

Binary search

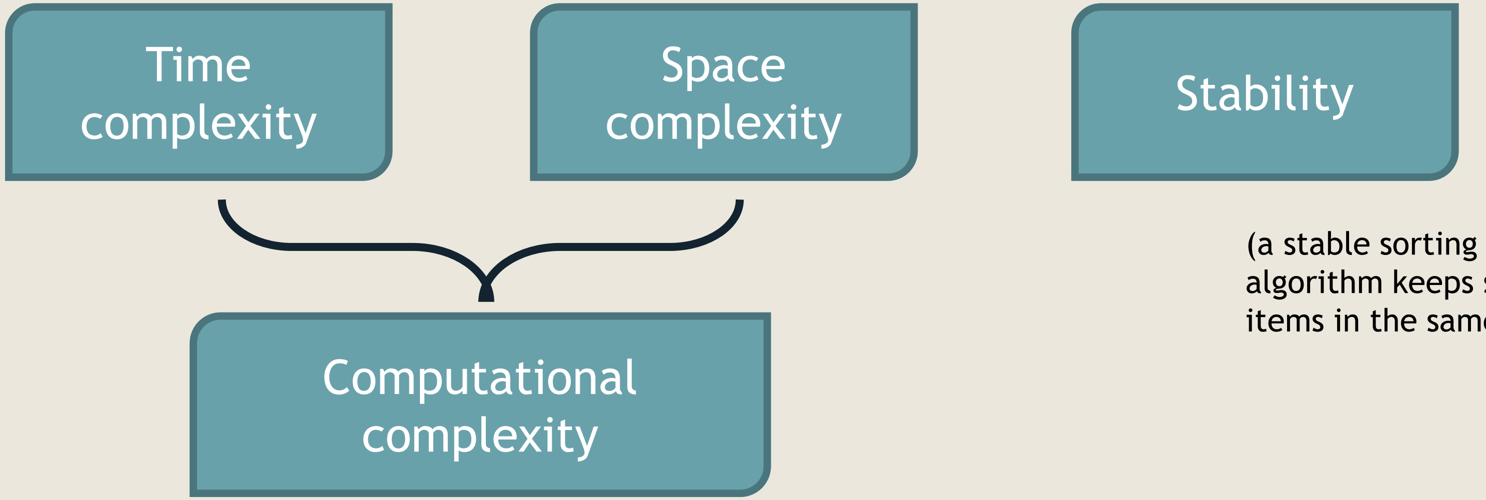
- Requires $\log_2 n$ steps on average**
- ceiling($\log_2 n$) steps at most
- List must be sorted
- More efficient – gap increases as n gets larger

*this could be reduced to n/2 on average if the list is known to be sorted
**$\log_2(n)=\log(n)/\log(2)$

Complexity of Algorithms

Comparing algorithms

- Let's assume we have two algorithms that both solve the same problem.
- Which is better?
  - *To answer, we need to measure the efficiency / performance of each.*
  - *Three factors*



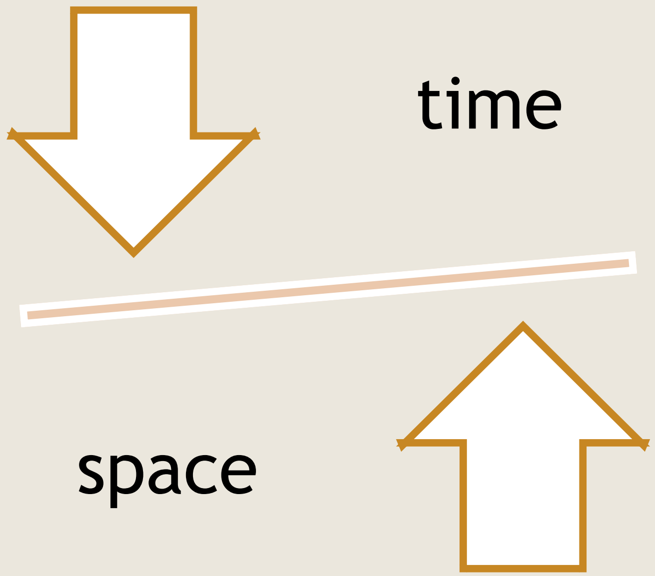
(a stable sorting algorithm keeps similar items in the same order)

Time complexity

- A measure of the time required
- Why is speed measured in elapsed time not always a good measure?
  - *varies with the computer used, other concurrent processes etc*
- What would be a better measure?
  - *Count the number of operations required*
  - *e.g. additions, multiplications, comparisons, bit swaps*
- Depends on the size of the input
  - *e.g. for sorting algorithms, how many items need to be sorted*
  - *We used no. of comparisons when comparing search algorithms*

Space complexity

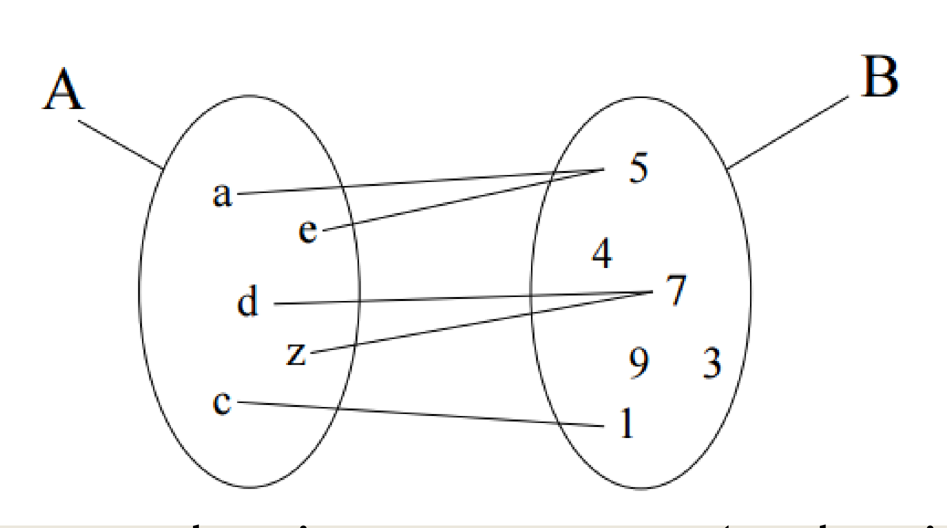
- The amount of temporary, additional memory required (RAM)
- Often depends on input size
  - *Not always – e.g. many sorting algorithms require constant amount of memory*
- Trade-off between time and space complexity
  - *More memory → faster*
- Space complexity often less important
  - *Except in memory-limited hardware – e.g. embedded systems*



- So, complexity of algorithms usually depends on the size of the input.
- We can say that the complexity is a function $f(n)$ where n is the number/size of input.
- How can we measure and compare the growth of functions?

Functions

Notation and terminology for functions



domain *maps to* **range (co-domain)**

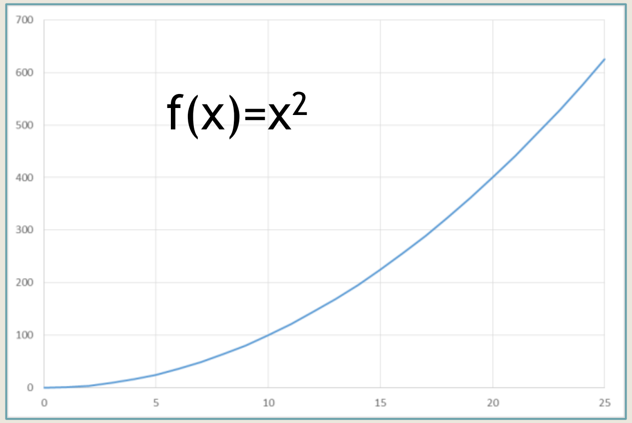
f* : *A* → *B

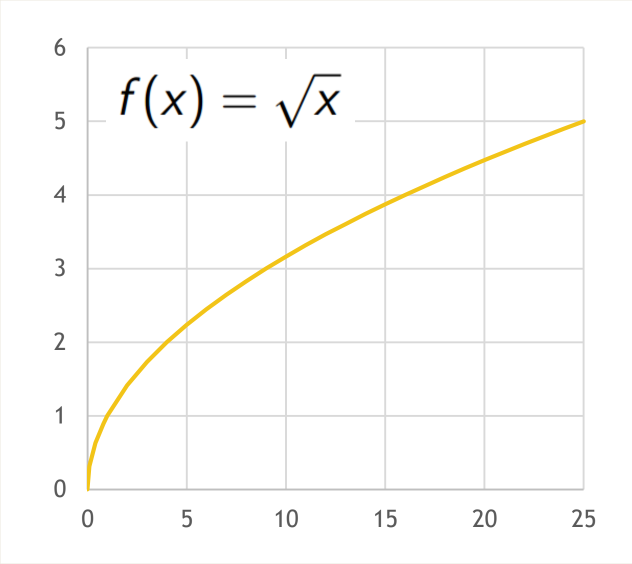
Here we have a **discrete** function f with only a few values:

f (a) = 5; f (e) = 5; f (d) = f (z)= 7; f (c) = 1

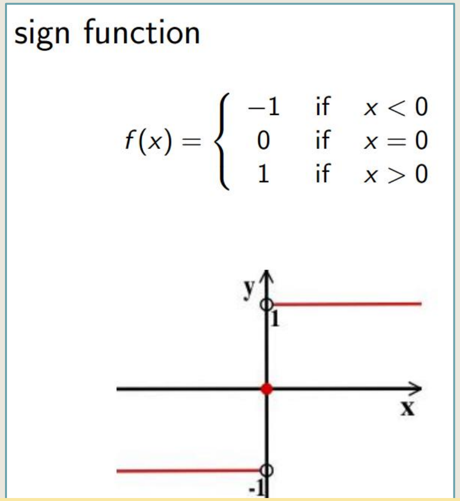
Continuous and piecewise functions

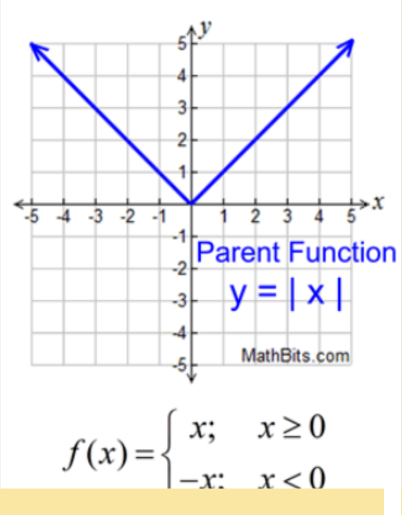
- The domain is a range of real numbers
- Continuous functions have no "break" (can draw line without lifting pen)



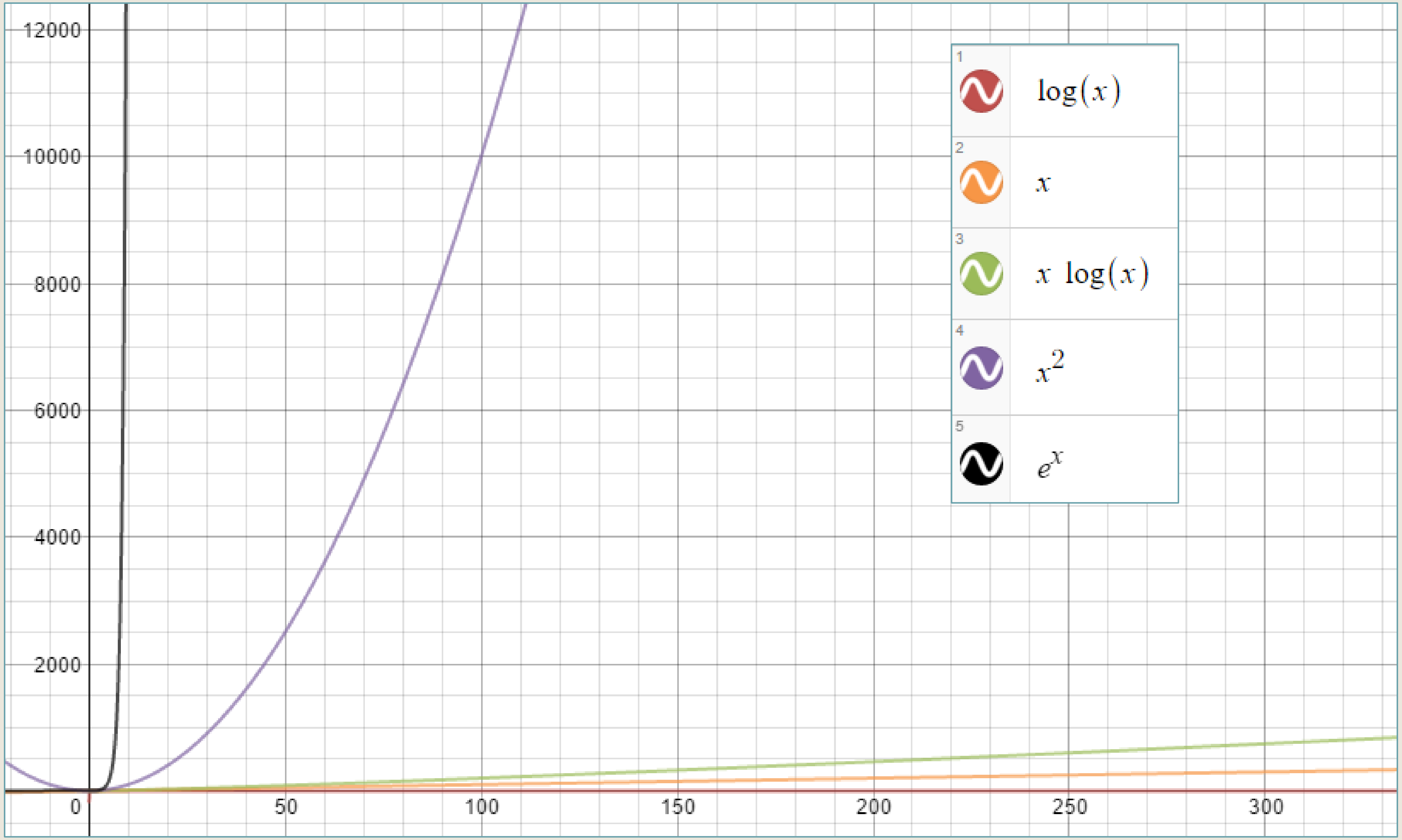


- Piecewise functions have several "pieces" (different behaviours)
- The absolute value function is both continuous and piecewise





Common functions compared



functions in Python

$$f(x) = \begin{cases} -1 & \text{if} \quad x < 0 \\ 0 & \text{if} \quad x = 0 \\ 1 & \text{if} \quad x > 0 \end{cases}$$

Defining the function

```
def sign_function(x):
      if x < 0:
            return -1
      elif x == 0:
            return 0
      else:
            return 1
```

Using the function

```
sign_function(8)
```

Asymptotic growth of functions

The BIG O

Asymptotic growth of functions

- Expressed using "Big-O" notation O(g(x))
- O(g(x)) describes the limiting behaviour of a function f(x) when the argument x grows without bounds (i.e. tends towards infinity)
- For comparison, we look for g(x) to be a simple function (x, x^2, log(x) etc)

Definition

We say a function f(x) is O(g(x))
if there exist two constants, C and k, such that

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$.

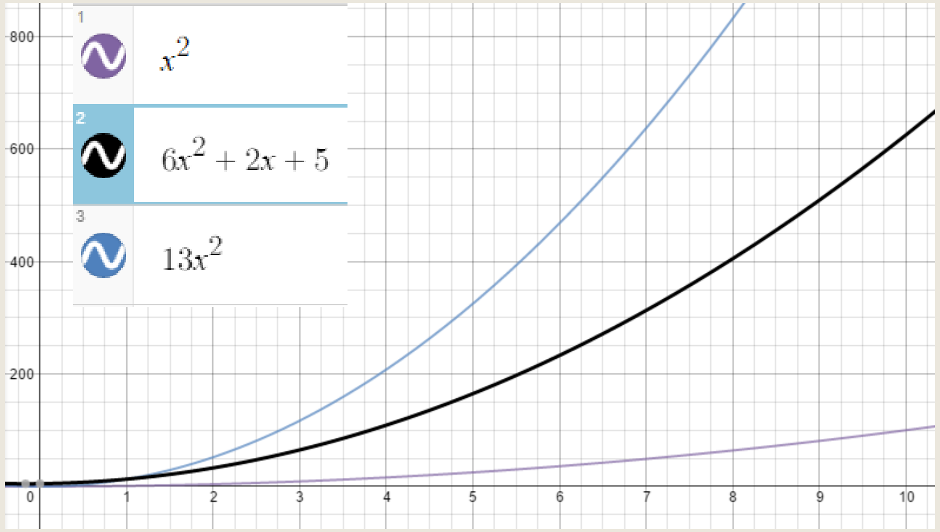
This is written as f(x) is O(g(x)) or $f(x) \epsilon$ O(g(x))

And pronounced "f(x) is big-oh of g(x)".

The constants C and k are called the "witnesses".

Example: Big-O of $f(x) = 6x^2 + 2x + 5$

- When x>1, $1<x^2$ and $x<x^2$
- Therefore when x>1,
  $6x^2 + 2x + 5$
  $< 6x^2 + 2x^2 + 5x^2$
  $= 13x^2$
- Thus
  $f(x) = 6x^2 + 2x + 5$ is $O(x^2)$



Simplification rules

To derive g(x) from f(x) so that f(x) is O(g(x)):

- If f(x) is a sum of several terms, keep only the term with the largest growth rate.
- If f(x) is a product of several factors, any constants (terms in the product that do not depend on x) can be omitted.

- Applying this to our example, $f(x) = 6x^2 + 2x + 5$,
  $6x^2$ is the term with the largest growth rate so f(x) is $O(6x^2)$
  6 is a constant, so f(x) is $O(x^2)$

Combinations of functions

If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$

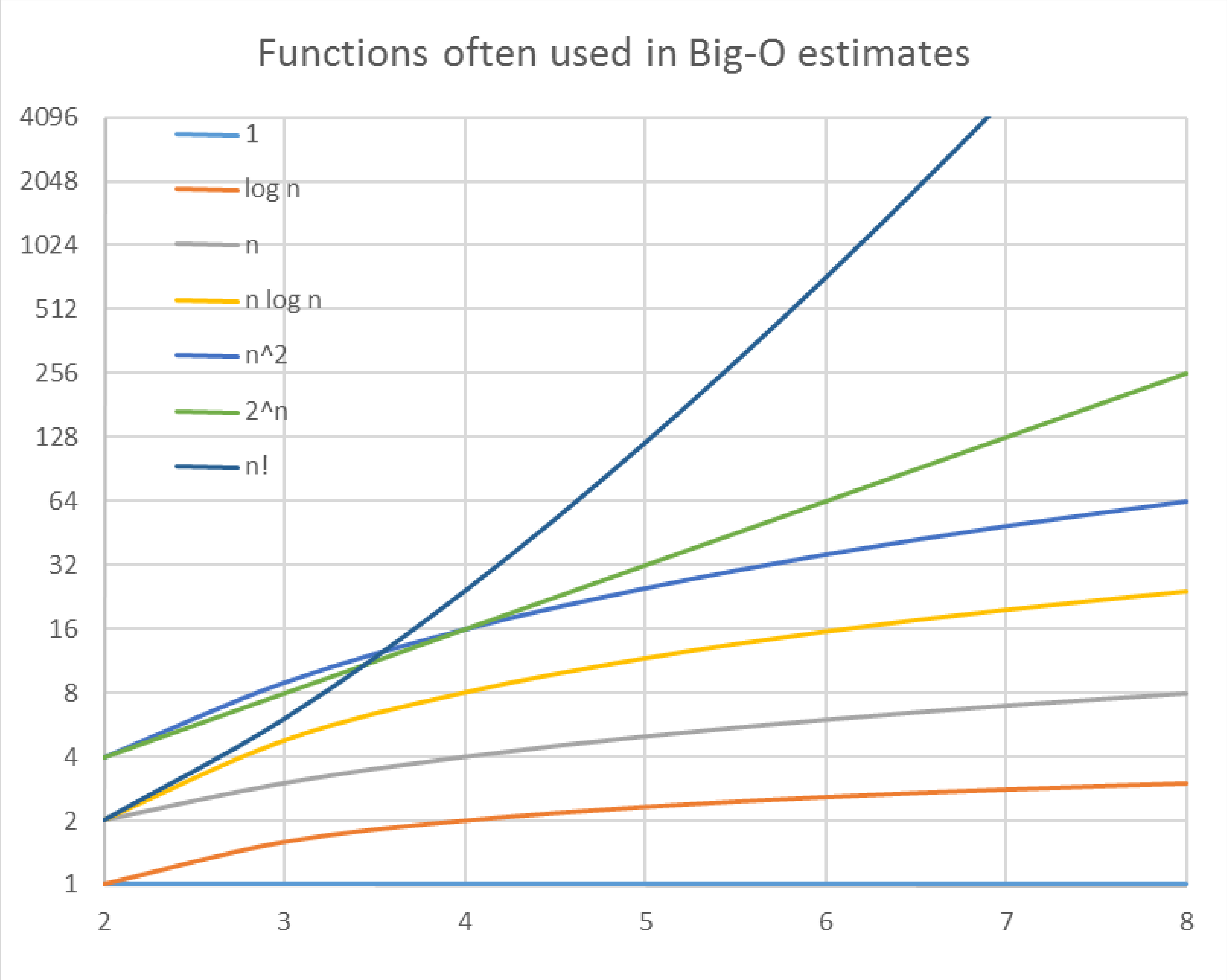
then

$(f_1 + f_2)(x)$ is $O(\max(g_1(x), g_2(x)))$

and

$(f_1 f_2)(x)$ is $O(g_1(x)\, g_2(x))$

This implies that
if $f_1(x)$ is $O(g(x))$ and $f_2(x)$ is also $O(g(x))$ then $(f_1 + f_2)(x)$ is $O(g(x))$


Functions often used in Big-O estimates
4096
2048
1024
512
256
128
64
32
16
8
4
2
1
2
3
4
5
6
7
8
1
log n
n
n log n
n^2
2^n
n!

Big- Θ (theta): when big-O is not enough

- O(x) gives an asymptotic **upper** bound
- It could be **any upper** bound
- for example,
  $f(x) = 6x^2 + 2x + 5$ is
  O(x!) and also $O(x^{99})$ and also $O(x^2)$

- Big-Omega, Ω(g(x)) can be used to find a **lower** bound
  - f(x) is Ω(g(x)) if there exist two constants, C and k, such that

$$|f(x)| \geq C|g(x)| \text{ whenever } x > k$$

- Θ(x) combines O and Ω and gives a **tight** asymptotic bound

The order of functions (big-Theta Θ)

f(x) is of order g(x) or f(x) is Θ(g(x))
if there exist three constants, C_1, C_2 and k, such that

$$C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|$$

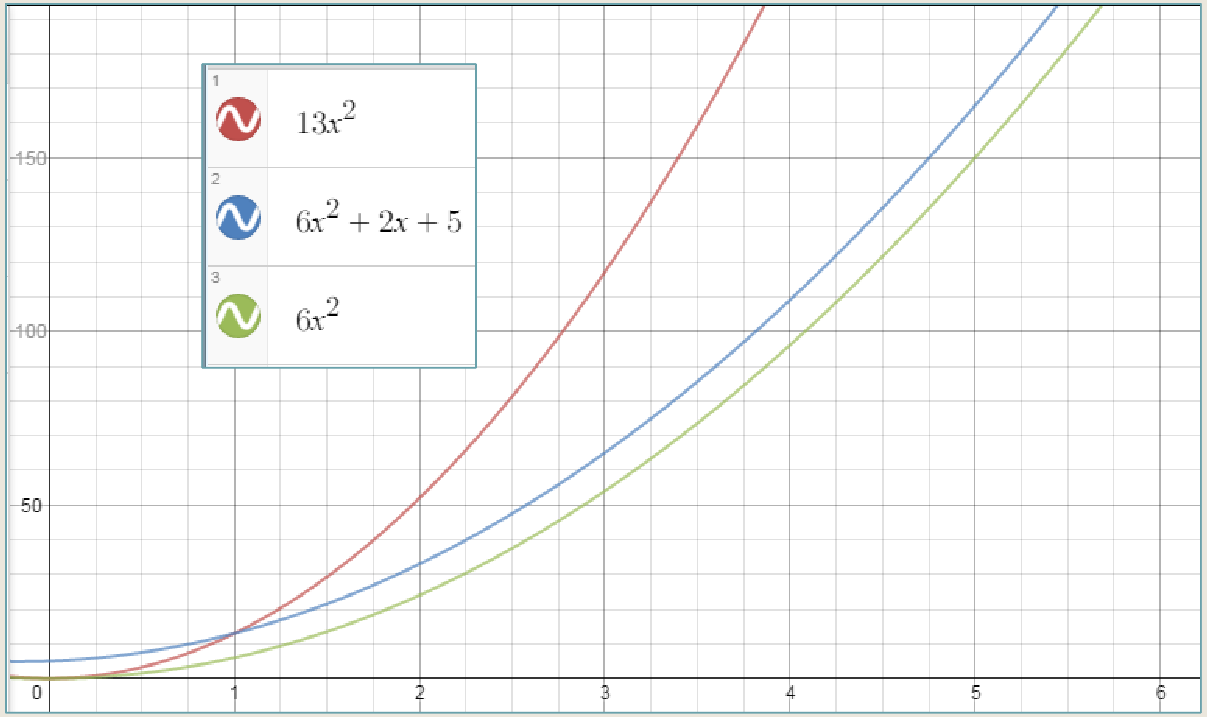
whenever $x > k$.

This means that

- Θ(x) gives a **tight** asymptotic bound
- f(x) is Θ(g(x)) if and only if f(x) is both Ω(g(x)) and O(g(x))
- If f(x) is Θ(g(x)) then g(x) is Θ(f(x))

Example: Big-Theta of $f(x) = 6x^2 + 2x + 5$

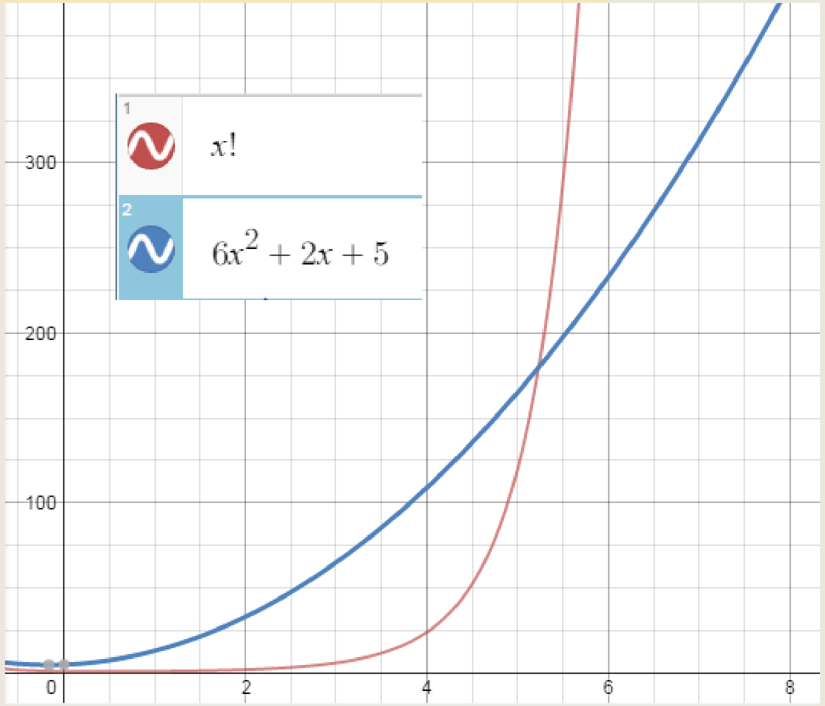
- We already know that $f(x) = 6x^2 + 2x + 5$ is $\mathbf{O(x^2)}$
- When x>1: 2x+5>0
  and thus
  $6x^2 + 2x + 5 > 6x^2$
- Thus $f(x) = 6x^2 + 2x + 5$ is $\mathbf{\Omega(x^2)}$
- So $f(x) = 6x^2 + 2x + 5$ is $\mathbf{\Theta(x^2)}$



- For all polynomials, the leading term (the one with the highest power) determines the order

Big-O and big-Theta compared

- Big-Theta is a **tight** bound
- much more informative than big-O, but:
  - *Θ can be hard or impossible to calculate*
  - *we usually give the lowest known upper bound as O anyway*
  - *In practice, they are almost interchangeable*
    (and many people say big-O when they mean "order of" i.e. big-Θ)



Application of Big-O to Algorithms

Application to algorithms

- For time complexity, estimate the number of "important" operations in terms of the size of the input
- For space complexity, estimate the additional temporary memory required
- Give Theta where possible
  - *Otherwise give the lowest known upper bound as big-O*

Complexity of Bubble Sort

```
Procedure bubblesort(a_1,..., a_n (n>=2))
    for i:=1 to n-1
      for j:=1 to n-i
          if a_j > a_{j+1}
             swap( a_j, a_{j+1} )
      stop if no swaps made for this j
```

Executes n-1, then n-2, then n-3, …, then 2 then 1 times

- Time complexity estimation: number of comparisons required (in terms of n)
- Total number of comparisons is exactly

$$(n-1)+(n-2)+\cdots+2+1=\frac{(n-1)n}{2}=\frac{1}{2}n^2-\frac{1}{2}n$$

- So Bubblesort is $\Theta(n^2)$
- This is the modified algorithm that stops if no swaps made but that usually doesn't make much difference

General rules for complexity of algorithms

- As a rule of thumb, the nested loops used in an algorithm determine the complexity

Number of nested loops over n (or n-1 etc)	Order of algorithm (rule of thumb)
No loops	$\Theta(1)$
One loop	$\Theta(n)$
Two loops	$\Theta(n^2)$

- A recursion that splits the list in half each time or similar is $\Theta(\log n)$ (e.g. binary search)
- Combining this, a recursion over half which includes a loop over n is $\Theta(n \log n)$
- A recursion that solves a problem of size N by recursively solving two smaller problems of size N-1 is $\Theta(2^n)$ (e.g. chicken nuggets)

Big-O of a^b mod m

exp1.py - F:/Dropbox/CSN08115_MADC/Petra_notes/exp1.py (3.6.1)

File Edit Format Run Options Window Help

```python
# "Obvious" algorithm
def power(a,b,m):
        acc = 1
        for i in range(b):
                acc = acc * a
        return acc % m

# Some test runs
print (power(3,4,100))
print (power(2,10,1000))
# A prime according to
# https://primes.utm.edu/lists/small/millions/
p = 941083987
# Fermat's Little Theorem says this will be 1
print (power(2,p-1,p))
```

Ln: 16

O(b)

exp2.py - F:/Dropbox/CSN08115_MADC/Petra_notes/exp2.py (3.6.1)

File Edit Format Run Options Window Help

```python
# Smarter algorithm
def power(a,b,m):
        if b==0:
                return 1
        sqrt = power(a,b//2,m)
        if b % 2 == 0:
                return (sqrt*sqrt) % m
        else:
                return (a*sqrt*sqrt) % m

# Some test runs
print (power(3,4,100))
print (power(2,10,1000))
# A prime according to
# https://primes.utm.edu/lists/small/millions/
p = 941083987
# Fermat's Little Theorem says this will be 1
print (power(2,p-1,p))
```

Ln: 19 Col: 0

O(log b)

Check this out with scripts exp1_complexity.py and exp2_complexity.py

What do typical big-O values mean for algorithms in practice?

[n is the problem size, e.g. the length of the array. Adapted from http://www.cs.cmu.edu/~mrmiller/15-121/Lectures/14-bigOh.pdf.]

O(1)	"Constant Time"	runtime does not depend on n		excellent	• Add to hash table • Retrieve from hash table
O(log n)	"Logarithmic Time"	runtime is proportional to log n	Doubling the problem size, runtime grows by a constant	good	• Binary search • Modular exponent (smart)
O(n)	"Linear Time"	runtime is proportional to n	Doubling the problem size, time doubles	fair	• Linear search • Lookup in an unsorted list

What do typical big-O values mean for algorithms in practice?

[n is the problem size, e.g. the length of the array. Adapted from http://www.cs.cmu.edu/~mrmiller/15-121/Lectures/14-bigOh.pdf.]

$O(n \log n)$	"log-linear Time"	runtime is proportional to n log n		bad	• Quick sort • Merge sort
$O(n^2)$	“Quadratic Time”	runtime is proportional to n^2	A linear time operation applied a linear number of times	horrible	• Bubble sort
$O(2^n)$	“Exponential Time”	runtime is proportional to 2^n	Add one to the problem size, runtime doubles	really horrible	• Guessing a password with n letters • Fibonacci series calculation

Scripting for Cybersec & Networks

What does Big-O tell you for algorithms?

- It does **not** tell you the numerical running time of algorithm for a particular input or for small n.
- It does tell you something about the rate of growth as the size of the input increases:
  - *At some point, an $O(n)$ algorithm will be faster than an $O(n^2)$ algorithm, always.*
  - *As the input size grows, the $O(n)$ algorithm will get increasingly faster than an $O(n^2)$ algorithm.*
  - *But cannot tell you for what values of n the $O(n)$ algorithm is faster than the $O(n^2)$ algorithm.*
  - *Similarly, an $O(n \log n)$ algorithm will get increasingly faster than an $O(n^2)$ algorithm.*

Worst case, average case, best case

- The performance of all algorithms will depend on the nature of the input
- For sorting algorithms, we could get very different results for lists that are sorted already, sorted in reverse order, nearly sorted, "random"
- We therefore often give 3 values: average case, worst case and best case
- If only one value given it is usually worst case
- Average case can be much more difficult to calculate than worst case

Bubble sort Worst case, average case, best case

- Best case: List is already sorted
  - *Algorithm will stop after first pass through the list, (n-1) comparisons*
  - *Complexity $\Theta(n)$*
- Worst case: List is sorted "the opposite way round"
  - *Algorithm needs all possible comparisons*
  - *Complexity $\Theta(n^2)$*
- Average case:
  - *Might stop one or two passes before the end, makes little difference, still $\Theta(n^2)$*

Bubble sort Space complexity

- Bubble sort needs only space for one temporary value (during the swap)
- So space complexity is $\Theta(1)$
- It couldn't be better!

Other sorting algorithms

Excellent | Good | Fair | Bad | Horrible

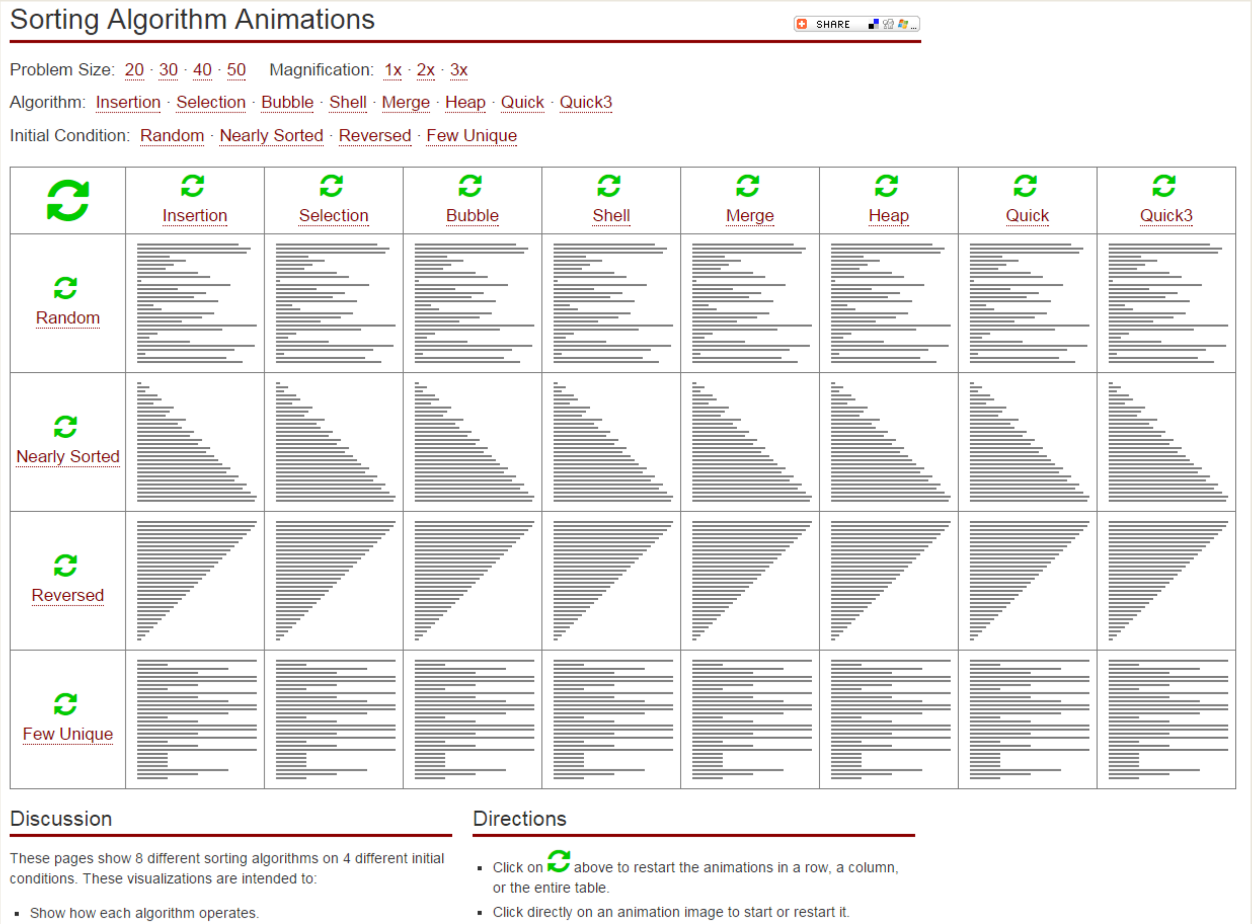
Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)

Used by Python

- Many of these are actually tight bounds Θ. O is often used instead, though formally this is less informative
- Tables of complexities: http://bigocheatsheet.com/ (table above) or https://en.wikipedia.org/wiki/Sorting_algorithm#Comparison_of_algorithms

But…

- In practice, the times taken may appear different
  - *For example, the constant factors ignored by big-O can make a difference*
- Note the trade-off between time and space complexity
- Some sorting algorithms are clearly very inefficient, but there is no single "best" sorting algorithm!
- See animations at http://www.sorting-algorithms.com/.



Algorithm Complexity Estimation with Python

Algorithm complexity estimation with Python

- exp1_complexity.py and exp2_complexity.py (moodle) show how you can **count** the number of times a function is executed within a program
- For large n, this empirical counter should be close to the theoretical O(n)
- But it is incremented in every loop so will take extra time
- An alternative is to **time** program execution

Timing code execution in Python

Timing code execution with Python

- Useful Python modules:
  - *time*
  - *timeit*
  - *cProfile*
  - *line_profiler (not part of the standard library)*

How long does my code take to run?

- Important for evaluating an application / development project
- helps determine which alternative is "better"

- **time** module → used in lab examples
  - *easy to use*
  - *Now contains functions that measure in nanoseconds*
- **timeit** module
  - *a bit clunky*
- cProfile and line_profiler
  - *Use to find out which parts of code are most "expensive"*

Timing your code with time.time

- Approach:
  - *Import time module*
  - *Record time before execution*
  - *Record time after execution*
  - *Subtract to get the elapsed time*

- Issues?
  - *Will vary depending on other processes running on machine*
  - *May be so short that it's difficult to see differences*

Hash tables

Hash Table example

The crux of Hash tables is the creation of an index.
To calculate the index (address) of a key, we use a hash function. In this example names are the keys; they are hashed on the first letter (that's the hash function here).

Andrew starts with "A" and so hashes to 0.

index

Address	Pointer
0	*
1	
2	*
3	
…	

Key	Andrew
Value	2753
Next	

The keys "Chris" and "Claire" collide - they have the same hash.
They form a chain.

Key	Claire
Value	2756
Next	*

Key	Chris
Value	2754
Next	

Names starting with "C" hash to 2.

Hash Table properties

- A hash table allows fast, random access to data.
- The hash table contains key and value pairs.
- Python **dictionary data type** implements hash tables (similar syntax in many other programming languages)
- massively useful
  - *Keys can be pretty much anything – usually strings*
  - *Values can be anything*
  - *Keys are not stored in order*
  - *Keys must be unique*
- Insert, access, delete, list keys – all fast
- Direct lookup by applying the hash function: searching is O(1)

Hash functions

- A hash function takes an input and returns a seemingly unrelated value within a small, known range (e.g. 0…99 for a hash table size 100)
- A typical hash function:
  - *Is fast*
  - *Is deterministic*
  - *Avoids clustering*
    (Having a hash key collision is not a show stopper – but long chains will result in poor performance).

Hash tables in Python

- A hash table is just a Python dictionary
  - *It uses an internal hash function behind the scenes*
  - *It's just a coincidence that we happened to use md5 hashes as keys in one of our dictionaries*
- The keys must be unique
- It's ok to have no values ("None")
  - *dict.fromkeys() converts a list of keys to a dictionary with "None" values*

```
>>> mylist=['a','b','cdr']
>>> mydict=dict.fromkeys(mylist)
>>> mydict
{'a': None, 'b': None, 'cdr': None}
>>> mydict2={key: None for key in mylist}
>>> mydict2
{'a': None, 'b': None, 'cdr': None}
```

Graphs in Python: pyplot

pyplot: Plotting graphs with Python

- There are several Python modules for plotting graphs
- E.g. pyplot (part of matplotlib)
  - *Tutorial* *https://matplotlib.org/tutorials/introductory/pyplot.html*
  - *more examples* *https://matplotlib.org/gallery/index.html*
- Remember you will need to
  pip install matplotlib
  (at windows command prompt)
  before you can import matplotlib.pyplot

pyplot Example (pyplot_ex1.py)

pyplot_ex1.py - F:\Dropbox\CSN08115_MADC\Petra_notes\pyplot_ex1.py (3.6.1)

File Edit Format Run Options Window Help

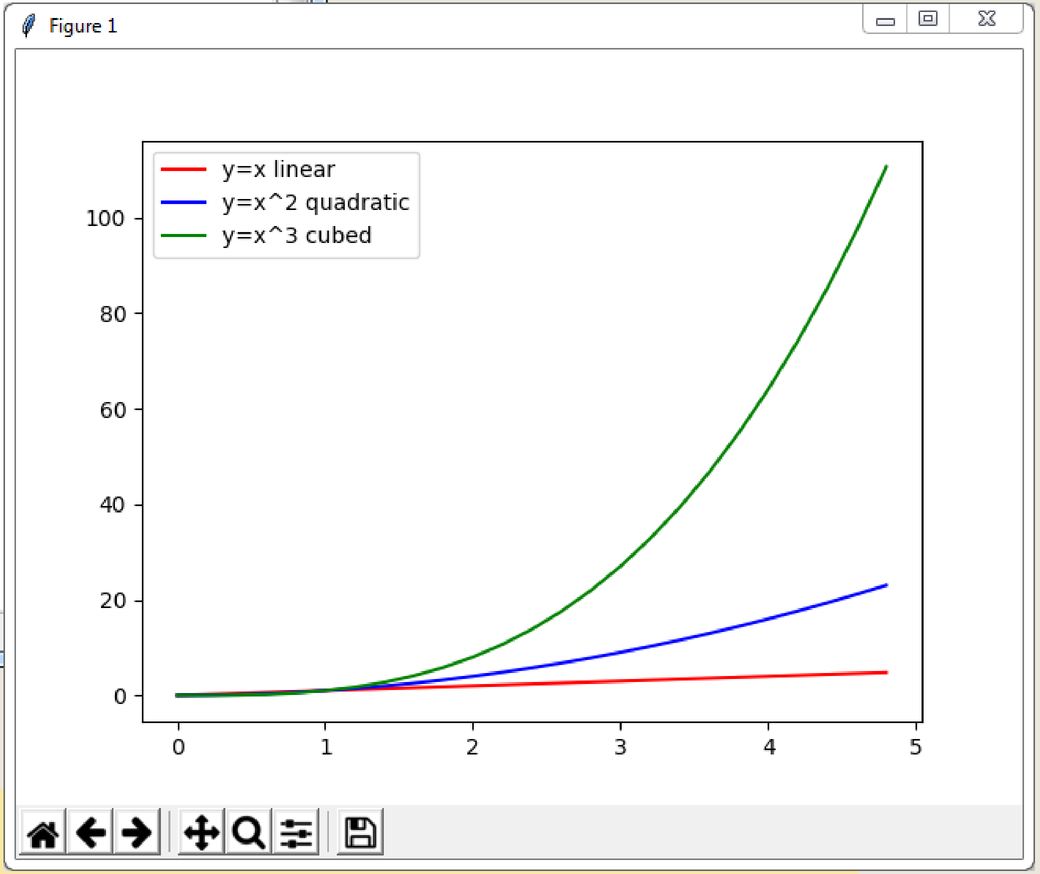
```python
# Example Pyplot Three from https://matplotlib.org/gallery/index.html

import matplotlib.pyplot as plt #(may need to pip install matplotlib)

# evenly sampled time at 200ms intervals
t = [i/5 for i in range(0,25)]
t_sq=[i**2 for i in t]
t_cube=[i**3 for i in t]

# red, blue and green lines with legends
plt.plot(t, t, 'r', label='y=x linear')
plt.plot(t, t_sq, 'b', label='y=x^2 quadratic')
plt.plot(t, t_cube, 'g', label='y=x^3 cubed')
plt.legend() # needed to actually create the legend

plt.show()
```



pyplot Example - dissected

```
plt.plot(t, t_sq)
plt.show()
```

plot() needs two lists,
one with x values (here t) and
one with y values (here t_sq)

show() is to
actually show
the plot

```
plt.plot(t, t_sq, 'b', label='y=x^2')
plt.legend()
plt.show()
```

Additional variables can be added to customise plot further, e.g. labels
To show the labels we need legend()

Optional third variable defines colour, line style etc
e.g.
'b' = blue (line)
'ro' = red circles
'g^-' = green triangles and line

pyplot Example using numpy (pyplot_numpy.py)

- numpy module allows us to
  - *create ranges with decimal steps*
  - *create arrays which have more functionality than lists (e.g. element-wise power)*

pyplot_numpy.py - F:/Dropbox/CSN08115_MADC/Petra_notes/pyplot_numpy.py (3.6.1)

File Edit Format Run Options Window Help

```python
# Example Pyplot Three from https://matplotlib.org/gallery/index.html
import numpy as np   #(may need to pip install numpy)
import matplotlib.pyplot as plt #(may need to pip install matplotlib)

# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2) # range() only works with integer steps,
                           # np.arange() allows decimal steps
                           # and creates an "ndarray" not a list

# red circles, blue squares with line and green triangles
plt.plot(t, t, 'ro', t, t**2, 'bs-', t, t**3, 'g^')
plt.show()
```

Ln: 13 Col: 0

Practical Lab 05

Some Resources

You should have a look at these links in your own time. Some are easier to understand than others, some more in depth - they are in a loose order.
The lab exercises will give links that are specifically useful for each exercise.

- https://www.youtube.com/watch?v=v4cd1O4zkGw
  (Big O notation intro video, 8 mins, sound/subtitles)
- https://en.wikipedia.org/wiki/Big_O_notation
- https://en.wikipedia.org/wiki/Analysis_of_algorithms
- https://www.cs.cmu.edu/~adamchik/15-121/lectures/Algorithmic%20Complexity/complexity.html
- http://www.sorting-algorithms.com/ - animations of sorting algorithms
- https://www.python-course.eu/python3_global_vs_local_variables.php - global variables in Python
- pyplot tutorial https://matplotlib.org/tutorials/introductory/pyplot.html
- https://matplotlib.org/gallery/index.html - more examples for plotting with Python
- https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation
- https://www.khanacademy.org/computing/computer-science/algorithms/sorting-algorithms/a/analysis-of-selection-sort
- https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/analysis-of-insertion-sort